Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №6

з дисципліни

«Алгоритми та структури даних»

Виконав: студент групи ІМ-42

Максим Крамаренко Юрійович

номер варіанту: 17

Перевірив:

Сергієнко А. М.

Київ 2025

Постановка задачі:

1. Представити зважений ненапрямлений граф із заданими параметрами так само, як у лабораторній роботі №3.

Відмінність 1: коефіцієнт k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.05. Отже, матриця суміжності Adir напрямленого графа за варіантом формується таким чином:
1) встановлюється параметр (seed) генератора випадкових чисел, рівне номеру варіанту n1n2n3n4;
2) матриця розміром n * n заповнюється згенерованими випадковими числами в діапазоні [0, 2.0);
3) обчислюється коефіцієнт k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.05, кожен елемент матриці множиться на коефіцієнт k;
4) елементи матриці округлюються: 0 — якщо елемент менший за 1.0, 1 — якщо елемент більший або дорівнює 1.0. Матриця Aundir ненапрямленого графа одержується з матриці Adir так само, як у ЛР №3.

Відмінність 2: матриця ваг W формується таким чином.
1) матриця B розміром n * n заповнюється згенерованими випадковими числами в діапазоні [0, 2.0) (параметр генератора випадкових чисел той же самий, n1n2n3n4);
2) одержується матриця C:
c(i,j) = ceil(b(i,j) * 100 * aundir(i,j)) c(i,j) Є C, b(i,j) Є B, Aundir(i,j) Є Aundir, де ceil — це функція, що округляє кожен елемент матриці до найближчого цілого числа, більшого чи рівного за дане;
3) одержується матриця D, у якій d(i,j) = 0, якщо c(i,j) = 0, d(i,j) = 1, якщо c(i,j) > 0, d(i,j) Є D, c(i,j) Є C;
4) одержується матриця H, у якій h(i,j) = 1, якщо d(i,j) ≠ d(j,i), та h(i,j) = 0 в іншому випадку;
5) Tr — верхня трикутна матриця з одиниць (tr(i,j) = 1 при i < j);
6) матриця ваг W симетрична, і її елементи одержуються за формулою: w(i,j) = w(j,i) = (d(i,j) + h(i,j) * tr(i,j)) * c(i,j).

2. Створити програму для знаходження мінімального кістяка за алгоритмом Краскала при n4 — парному і за алгоритмом Пріма — при непарному. При цьому у програмі:
- графи представляти у вигляді динамічних списків, обхід графа, додавання, віднімання вершин, ребер виконувати як функції з вершинами відповідних списків;
- у програмі виконання обходу відображати покроково, черговий крок виконувати за натисканням кнопки у вікні або на клавіатурі.

3. Під час обходу графа побудувати дерево його кістяка. У програмі дерево кістяка виводити покроково у процесі виконання алгоритму. Це можна виконати одним із двох способів:
- або виділяти іншим кольором ребра графа;
- або будувати кістяк поряд із графом.
При зображенні як графа, так і його кістяка, вказати ваги ребер.

При проєктуванні програми також слід врахувати наступне:
1) мова програмування обирається студентом самостійно;
2) графічне зображення усіх графів має формуватися програмою з тими ж вимогами, як у ЛР №3;
3) всі графи обов'язково зображувати у графічному вікні;
4) типи та структури даних для внутрішнього представлення всіх даних у програмі слід вибрати самостійно.

Варіант 17:

n1 = 4
n2 = 2
n3 = 1
n4 = 7

SEED = 4217

n = n3 + 10

Текст програми:

Файл №1 (main.py**)**

```
from graph_utils import *
from matrix_utils import *

n1 = 4
n2 = 2
n3 = 1
n4 = 7

n = n3 + 10
```

```
k = 1.0 - n3 * 0.01 - n4 * 0.005 - 0.05

dir = get_dir(n, k)

undir = get_undir(dir)
print_matrix(undir, "Undirected Graph")

B = get_B(n)

C = get_C(undir, B)

D = get_D(C)

H = get_H(D)

W = get_W(C, D, H)
print_matrix(W, "W Matrix", 3)

def get_MinimumSpanningTree(W):
    n = len(W)
    in_mst = [False] * n
    MST = list()

    in_mst[0] = True

    for _ in range(n-1):
        min_weight = math.inf
        min_edge = (-1, -1)

        for u in range(n):
            if in_mst[u]:
                for v in range(n):
                    if not in_mst[v] and W[u][v] != math.inf and W[u][v] <
min_weight:
                        min_weight = W[u][v]
                        min_edge = (u, v)

        if min_edge[0] != -1:
            MST.append(min_edge)
            in_mst[min_edge[1]] = True
```

```
    return MST

MST = get_MinimumSpanningTree(W)

def get_weight(W, MST):
    weight = 0
    for edge in MST:
        weight += W[edge[0]][edge[1]]
    return weight

total_weight = get_weight(W, MST)

print("Minimum Spanning Tree Edges:")
for edge in MST:
    print(f"{edge[0] + 1} - {edge[1] + 1}")
print()

print(f"Total Weight of Minimum Spanning Tree: {total_weight}")

draw_graph(undir, directed=False, title="Undirected Graph", weight_matrix=W,
spanning_tree=MST)
```

Файл №2 (graph_utils.py**)**

```
import math
import random
import matplotlib.pyplot as plt
from matplotlib.widgets import Button

random.seed(4217)

def draw_graph(matrix, directed=False, title=None, weight_matrix=None,
spanning_tree=None):
    R = 10  # Radius of the circular layout
    angle_step = 2 * math.pi / (len(matrix) - 1) # Angle between nodes
    # Calculate node positions
    positions = []
    for i in range(len(matrix)):
        if i == 0:
            positions.append((0, 0))  # Center node
```

```
            continue
        angle = i * angle_step
        x = R * math.cos(angle)
        y = R * math.sin(angle)
        positions.append((x, y))

    node_R = 0.8  # Node radius

    # Determine graph type - simplified to only MST or normal
    graph_type = "normal"
    if spanning_tree:
        graph_type = "mst"

    # Helper function to adjust for node boundary
    def adjust_for_R(x1, y1, x2, y2, offset):
        dx, dy = x2 - x1, y2 - y1
        length = math.sqrt(dx ** 2 + dy ** 2)
        if length == 0:
            return x1, y1, x2, y2
        scale = (length - offset) / length
        return x1 + dx * (1 - scale), y1 + dy * (1 - scale), x2 - dx * (1 -
scale), y2 - dy * (1 - scale)

    # Create step-by-step visualization for MST if spanning_tree is provided
    steps = []
    if graph_type == "mst" and spanning_tree:
        mst_edges = []
        for i, edge in enumerate(spanning_tree):
            mst_edges.append(edge)
            steps.append({
                "mst_edges": mst_edges.copy(),
                "current_edge": edge
            })

    # Function to draw a node
    def draw_node(i, x, y, color):
        plt.scatter(x, y, s=500, color=color, edgecolor="black", linewidth=1,
zorder=2)
        plt.text(x, y, str(i + 1), fontsize=12, ha="center", va="center",
zorder=4)
```

```
    # Completely revise the MST step visualization approach
    if graph_type == "mst" and steps:
        # Create a persistent figure with two axes
        # One for the graph and one for the buttons
        fig = plt.figure(figsize=(8, 8))
        plt.subplots_adjust(bottom=0.2)  # Make space for buttons at the bottom

        # Create axes for buttons
        ax_prev = plt.axes([0.3, 0.05, 0.15, 0.075])
        ax_next = plt.axes([0.55, 0.05, 0.15, 0.075])

        # Create separate axes for graph content
        graph_ax = plt.axes([0.1, 0.2, 0.8, 0.7])

        # Keep track of current step
        current_step_index = [0]  # Use a list for mutable reference

        # Function to draw the step in the graph_ax
        def draw_mst_step(step_index):
            graph_ax.clear()

            step = steps[step_index]
            mst_edges = step.get("mst_edges", []) # Assumed to be 0-indexed
tuples, e.g., [(0, 1), (1, 2)]
            current_edge = step.get("current_edge") # Assumed to be a 0-indexed
tuple, e.g., (0, 1)

            graph_ax.set_title(f"{title or 'Minimum Spanning Tree'} - Step
{step_index + 1}/{len(steps)}", fontsize=14)

            # Plot nodes
            for i, (x, y) in enumerate(positions): # i is 0-indexed (0 to n-1)
                # Check if this node (index i) is part of the current edge
(0-indexed tuple)
                is_current_node = current_edge and (i == current_edge[0] or i ==
current_edge[1])

                # Check if this node (index i) is part of any MST edge so far
(0-indexed tuples)
                is_mst_node = any((i == edge[0] or i == edge[1]) for edge in
mst_edges)
```

```
                if is_current_node:
                    color = "#FF6347"  # Tomato for nodes in current edge
                elif is_mst_node:
                    color = "#FFD700"  # Gold for nodes in MST
                else:
                    color = 'lightgray'  # Unvisited nodes

                graph_ax.scatter(x, y, s=500, color=color, edgecolor="black",
linewidth=1, zorder=2)
                graph_ax.text(x, y, str(i + 1), fontsize=12, ha="center",
va="center", zorder=4)

            # Plot all edges
            for i in range(len(matrix)):
                for j in range(len(matrix)):
                    if matrix[i][j] and (directed or i <= j):
                        x1, y1 = positions[i]
                        x2, y2 = positions[j]

                        edge_in_mst = any(
                            ((i == e[0] and j == e[1]) or (i == e[1] and j ==
e[0]))
                            for e in mst_edges
                        )

                        is_current_edge = current_edge and (
                            (i == current_edge[0] and j == current_edge[1]) or
                            (i == current_edge[1] and j == current_edge[0])
                        )

                        if is_current_edge:
                            edge_color = "#FF6347"  # Tomato for the current
edge
                            edge_width = 3.0
                        elif edge_in_mst:
                            edge_color = "#FFD700"  # Gold for MST edges
                            edge_width = 2.5
                        else:
                            edge_color = "lightgray"  # Light gray for non-MST
edges
```

```
                            edge_width = 1.0

                        # Draw edge on graph_ax using 0-based i, j for positions
and weights
                        dx, dy = x2 - x1, y2 - y1
                        length = math.sqrt(dx ** 2 + dy ** 2)
                        x1_adj, y1_adj, x2_adj, y2_adj = adjust_for_R(x1, y1,
x2, y2, node_R)

                        weight_label = ""
                        if weight_matrix is not None and weight_matrix[i][j] !=
0:
                            weight_label = str(weight_matrix[i][j])

                        # If edge would pass through center, draw curved edge
                        if (length >= 2*(R - node_R) and length <= 2*(R +
node_R)):
                            norm_dx, norm_dy = dx / length, dy / length
                            perp_x, perp_y = -norm_dy, norm_dx
                            curve_factor = 3.0 + random.uniform(-0.5, 0.5)
                            midx = (x1 + x2) / 2 + perp_x * curve_factor
                            midy = (y1 + y2) / 2 + perp_y * curve_factor
                            t_values = [0, 0.25, 0.5, 0.75, 1.0]
                            curve_x = []
                            curve_y = []

                            for t in t_values:
                                bx = (1-t)**2 * x1_adj + 2*(1-t)*t * midx + t**2
* x2_adj
                                by = (1-t)**2 * y1_adj + 2*(1-t)*t * midy + t**2
* y2_adj
                                curve_x.append(bx)
                                curve_y.append(by)

                            graph_ax.plot(curve_x, curve_y, color=edge_color,
linewidth=edge_width, zorder=1)

                            # Add weight label for curved edge
                            if weight_label:
                                mid_t = 0.5
```

```
                                label_x = (1-mid_t)**2 * x1_adj +
2*(1-mid_t)*mid_t * midx + mid_t**2 * x2_adj
                                label_y = (1-mid_t)**2 * y1_adj +
2*(1-mid_t)*mid_t * midy + mid_t**2 * y2_adj
                                graph_ax.text(label_x, label_y, weight_label,
fontsize=10, ha="center", va="center",
                                              bbox=dict(facecolor=edge_color,
alpha=0.6, edgecolor=edge_color), zorder=5)
                        else:
                            # Draw straight edge
                            graph_ax.plot([x1_adj, x2_adj], [y1_adj, y2_adj],
color=edge_color, linewidth=edge_width, zorder=1)

                            # Add weight label for straight edge
                            if weight_label:
                                label_x = (x1_adj + x2_adj) / 2
                                label_y = (y1_adj + y2_adj) / 2
                                graph_ax.text(label_x, label_y, weight_label,
fontsize=10, ha="center", va="center",
                                              bbox=dict(facecolor=edge_color,
alpha=0.7, edgecolor=edge_color), zorder=5)

            graph_ax.set_xlim(-15, 15)
            graph_ax.set_ylim(-15, 15)
            graph_ax.axis("off")
            fig.canvas.draw_idle()  # Update the figure

        # Define button click handlers using partial functions for proper
binding
        def on_prev(event):
            if current_step_index[0] > 0:
                current_step_index[0] -= 1
                draw_mst_step(current_step_index[0])

        def on_next(event):
            if current_step_index[0] < len(steps) - 1:
                current_step_index[0] += 1
                draw_mst_step(current_step_index[0])

        # Create buttons with persistent event handlers
        btn_prev = Button(ax_prev, 'Previous')
```

```
        btn_next = Button(ax_next, 'Next')

        btn_prev.on_clicked(on_prev)
        btn_next.on_clicked(on_next)

        # Draw the initial step
        draw_mst_step(0)

        plt.show()
        return

    # If no steps for MST, proceed with normal graph drawing
    plt.figure(figsize=(8, 8))
    for i, (x, y) in enumerate(positions):
        draw_node(i, x, y, 'lightgray')

    # Draw edges
    for i in range(len(matrix)):
        for j in range(len(matrix)):
            # First check for self-loops to handle them specially
            if matrix[i][j] and i == j:
                edge_color = (random.randint(0, 235) / 255, random.randint(0,
235) / 255, random.randint(0, 235) / 255)
                # Draw self-loop for a node that connects to itself
                x, y = positions[i]
                loop_radius = 1
                if(x != 0 and y != 0):
                    vector_length = math.sqrt(x ** 2 + y ** 2)
                    x_loop = x + x * loop_radius / vector_length
                    y_loop = y + y * loop_radius / vector_length
                else:
                    x_loop = x
                    y_loop = y + loop_radius
                loop = plt.Circle((x_loop, y_loop), loop_radius,
color=edge_color, fill=False, zorder=1)
                plt.gca().add_patch(loop)
                 # Add weight label for self-loop (optional, near the loop)
                if weight_matrix is not None and weight_matrix[i][j] != 0:
                    weight_label = str(weight_matrix[i][j])
                    plt.text(x_loop, y_loop + loop_radius + 0.2, weight_label,
fontsize=10, ha="center", va="bottom",
```

```
                            bbox=dict(facecolor=edge_color, alpha=0.7,
edgecolor=edge_color), zorder=5)

            # Then handle regular edges (for undirected, only process each edge
once)
            elif matrix[i][j] and (directed or i < j):
                edge_color = (random.randint(0, 235) / 255, random.randint(0,
235) / 255, random.randint(0, 235) / 255)

                x1, y1 = positions[i]
                x2, y2 = positions[j]

                # Inline edge drawing logic starts here
                dx, dy = x2 - x1, y2 - y1
                length = math.sqrt(dx ** 2 + dy ** 2)
                x1_adj, y1_adj, x2_adj, y2_adj = adjust_for_R(x1, y1, x2, y2,
node_R)

                weight_label = ""
                if weight_matrix is not None and weight_matrix[i][j] != 0:
                    weight_label = str(weight_matrix[i][j])

                # Draw edges that would pass through center (curved)
                if (length >= 2*(R - node_R) and length <= 2*(R + node_R)):
                    norm_dx, norm_dy = dx / length, dy / length
                    perp_x, perp_y = -norm_dy, norm_dx
                    curve_factor = 3.0 + random.uniform(-0.5, 0.5)
                    midx = (x1_adj + x2_adj) / 2 + perp_x * curve_factor
                    midy = (y1_adj + y2_adj) / 2 + perp_y * curve_factor
                    t_values = [0, 0.25, 0.5, 0.75, 1.0]
                    curve_x = []
                    curve_y = []

                    for t in t_values:
                        bx = (1-t)**2 * x1_adj + 2*(1-t)*t * midx + t**2 *
x2_adj
                        by = (1-t)**2 * y1_adj + 2*(1-t)*t * midy + t**2 *
y2_adj
                        curve_x.append(bx)
                        curve_y.append(by)
```

```
                    if directed:
                        plt.plot(curve_x[:-1], curve_y[:-1], color=edge_color,
linewidth=1.0, zorder=1)
                        last_segment_x = curve_x[-2]
                        last_segment_y = curve_y[-2]
                        plt.arrow(last_segment_x, last_segment_y,
                                curve_x[-1] - last_segment_x,
                                curve_y[-1] - last_segment_y,
                                head_width=0.30, length_includes_head=True,
                                color=edge_color, linewidth=1.0, zorder=4)
                    else:
                        plt.plot(curve_x, curve_y, color=edge_color,
linewidth=1.0, zorder=1)

                    # Add weight label for curved edge
                    if weight_label:
                        mid_t = 0.5
                        label_x = (1-mid_t)**2 * x1_adj + 2*(1-mid_t)*mid_t *
midx + mid_t**2 * x2_adj
                        label_y = (1-mid_t)**2 * y1_adj + 2*(1-mid_t)*mid_t *
midy + mid_t**2 * y2_adj
                        plt.text(label_x, label_y, weight_label, fontsize=10,
ha="center", va="center",
                                bbox=dict(facecolor=edge_color, alpha=0.6,
edgecolor=edge_color), zorder=5)

                # Draw direct edges (straight)
                else:
                    if directed:
                        plt.arrow(x1_adj, y1_adj, x2_adj - x1_adj, y2_adj -
y1_adj, head_width=0.30, length_includes_head=True,
                                color=edge_color, linewidth=1.0, zorder=4)
                    else:
                        plt.plot([x1_adj, x2_adj], [y1_adj, y2_adj],
color=edge_color, linewidth=1.0, zorder=1)

                    # Add weight label for straight edge
                    if weight_label:
                        label_x = (x1_adj + x2_adj) / 2
                        label_y = (y1_adj + y2_adj) / 2
```

```
                        plt.text(label_x, label_y, weight_label, fontsize=10,
ha="center", va="center",
                                 bbox=dict(facecolor=edge_color, alpha=0.7,
edgecolor=edge_color), zorder=5)


    plt.xlim(-1,5*R, 1.5*R)
    plt.ylim(-1.5*R, 1.5*R)
    plt.axis("off")
    plt.show()
```

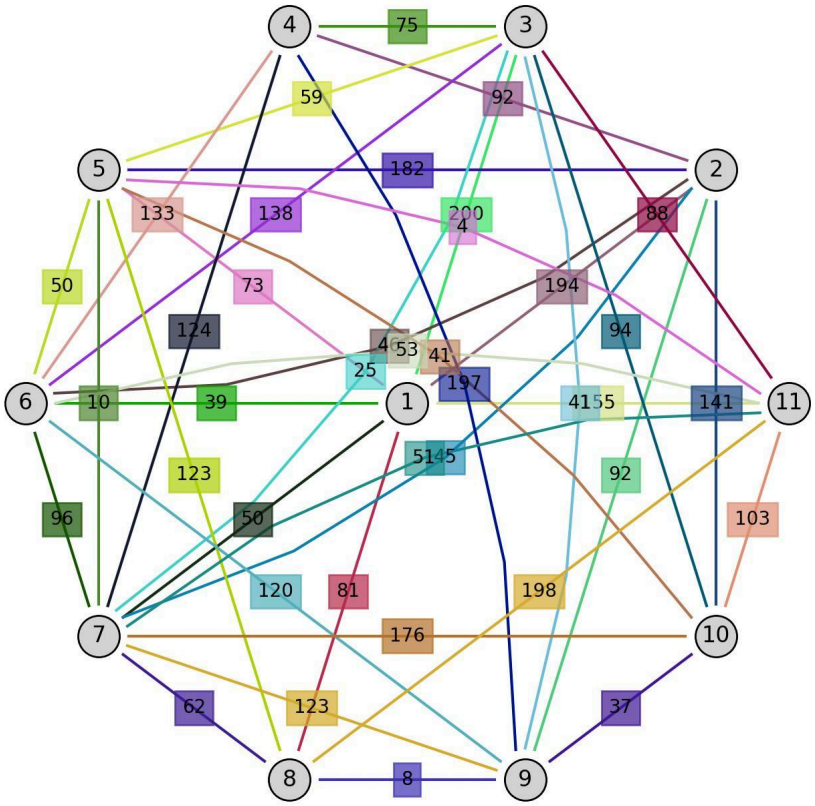
Тестування програми:

Матриця суміжності

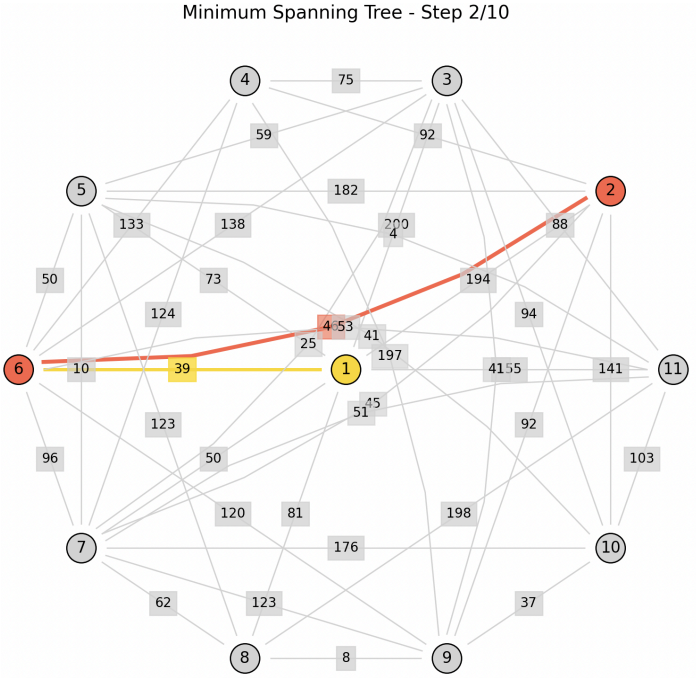
```
=== Undirected Graph ===
0 1 1 0 1 1 1 1 0 0 1
1 0 0 1 1 1 1 0 1 1 0
1 0 1 1 1 1 1 0 1 1 1
0 1 1 0 0 1 1 0 1 0 0
1 1 1 0 1 1 1 1 0 1 1
1 1 1 1 1 1 1 0 1 0 1
1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 1 0 1 1 1 0 1
0 1 1 1 0 1 1 1 1 1 0
0 1 1 0 1 0 1 0 1 0 1
1 0 1 0 1 1 1 1 0 1 0
```

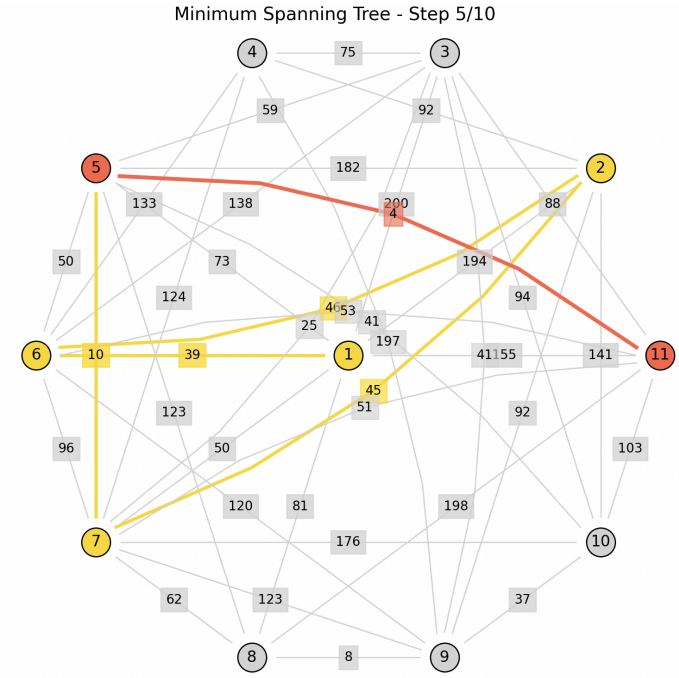
Матриця ваг

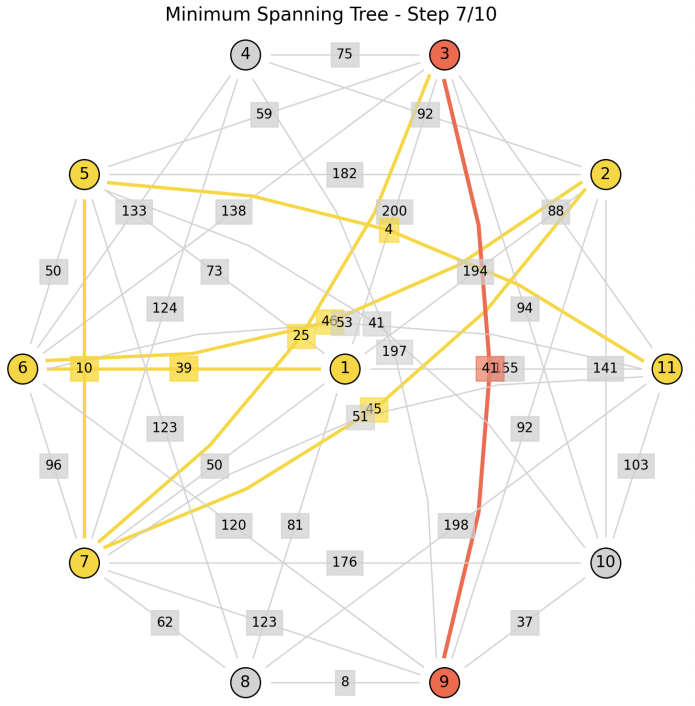
```
=== W Matrix ===
  0 194 200 inf  73  39  50  81 inf inf 155
194   0 inf  92 182  46  45 inf  92 141 inf
200 inf   0  75  59 138  25 inf  41  94  88
inf  92  75   0 inf 133 124 inf 197 inf inf
 73 182  59 inf   0  50  10 123 inf  41   4
 39  46 138 133  50   0  96 inf 120 inf  53
 50  45  25 124  10  96   0  62 123 176  51
 81 inf inf inf 123 inf  62   0   8 inf 198
inf  92  41 197 inf 120 123   8   0  37 inf
inf 141  94 inf  41 inf 176 inf  37   0 103
155 inf  88 inf   4  53  51 198 inf 103   0
```

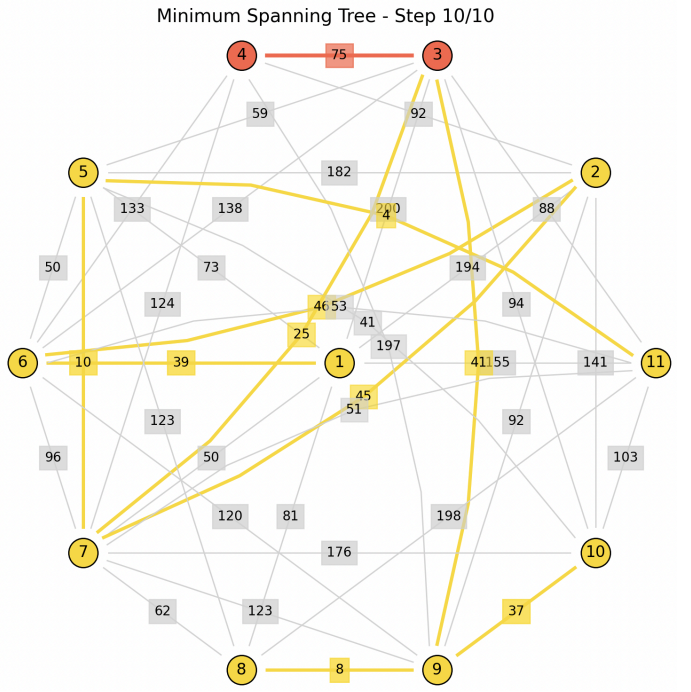


Мінімальний кістяк









Сума ваг ребер знайденого мінімального кістяка: 330

Висновок:

Засвоїв теоретичний матеріал лекцій навчився створювати мінімальні кістяки для графів, використовуючи алгоритм Пріма. Покращив практичний досвід у створенні функцій обходу графів та їх аналізуванні. Покращив навички роботи з графічними вікнами та функціями для них. Збільшив досвід роботи з матрицями та створення функцій для їх аналізу.